Graph-based data preparation for detecting buffer overflow vulnerabilities in code within CI/CD pipelines*

Oleg Savenko[1,†], Silvia Lips[2,†], Piotr Gaj[3,†] and Yevhenii Sierhieiev[1,*,†]

[1] Khmelnytskyi National University, Instytuts'ka St, 11, Khmelnytskyi, 29000, Ukraine
[2] Tallinna Tehhnikaülikool, Ehitajate tee 5, Tallinn, 12616, Estonia
[3] Silesian University of Technology, ul. Akademicka 2A, 44-100 Gliwice, Poland

Abstract

Buffer overflows remain among the most dangerous vulnerability classes in system and embedded software because they corrupt memory invariants, enable arbitrary control-flow transfers, and undermine critical components. We present a fully reproducible "data-to-vision" pipeline for automated detection of stack and heap overflows and off-by-one errors that combines formal risk conditions with graph-based program representations (AST/CFG/DFG) and multi-channel renders used to train a three-class detector with class-aware refinement. The core of the method is twofold. First, we introduce the notion of effective buffer capacity that deducts protocol-specific overheads and safety reserves from nominal allocation, aligning the decision boundary with what is actually safe to copy. Second, we define nodal (local) and path-level risk indicators that couple transfer estimates with guard signals (boundary checks, canaries) and off-by-one cues, thereby reducing false positives while preserving auditability at the level of minimal root-cause subgraphs. The pipeline operates as follows: from a fixed code snapshot and a stabilized preprocessor profile we construct a unified program graph that fuses control- and data-dependencies; we annotate buffers, sources/sinks, format-string and loop invariants; we compute edge-level transfer estimates and local/chain risks; candidate subgraphs are rasterized into multi-channel frames and labeled into {Stack, Heap, Off-by-one}, with curated hard negatives to improve specificity. All artifacts (schemas, toolchain, seeds, profiles) are version-locked and shipped in an OCI container, yielding byte-for-byte reproducibility in CI/CD and enabling SARIF outputs and blocking thresholds. On real-world corpora built from CVE/NVD cases and industrial examples with project-wise 70/15/15 splits, the approach consistently outperforms rule-only SAST baselines (e.g., Cppcheck, Flawfinder) and non-graph vision baselines in both detection quality and localization fidelity, while maintaining interpretable reports (class, score, code-span, matched template, explanation). The contributions are: (i) a deterministic data-preparation stack that turns program graphs into vision-ready inputs; (ii) formal, class-aware risk metrics that couple transfer size with effective capacity and guard signals; and (iii) a labeling and hard-negative strategy compatible with CI/CD evaluation without project leakage. Future extensions include other memory-safety classes (integer overflow, use-after-free, races), stronger XAI components (contrastive, counterfactual explanations), and code-span-level remediation suggestions tightly integrated into development workflows.

Keywords

cybersecurity, buffer overflow, machine learning, graphs, yolo, system software

1. Introduction

Software-driven digital transformation has expanded the attack surface of safety-critical domains (aviation, rail, energy, telecom), where a single memory-safety defect can cascade into service disruption or remote code execution. Despite the progress of SAST tools, buffer overflows (stack, heap, off-by-one) remain among the most impactful classes because they emerge from subtle interactions of control- and data-flow, preprocessor configurations, and build profiles. In high-

* AdvAIT-2025: 2nd International Workshop on Advanced Applied Information Technologies: AI & DSS, December 05, 2025, Khmelnytskyi, Ukraine, Zilina, Slovakia
[1,*] Corresponding author.
[†] These authors contributed equally.
savenko_oleg_st@ukr.net (O. Savenko); silvia.lips@taltech.ee (S. Lips); piotr.gaj@polsl.pl (P. Gaj); ysierhieiev@gmail.com (Ye. Sierhieiev)
0000-0002-4104-745X (O. Savenko); 0000-0001-7352-5965 (S. Lips); 0000-0002-2291-7341 (P. Gaj); 0009-0008-9877-9863 (Ye. Sierhieiev)

velocity CI/CD environments, organizations need reproducible, automatable pipelines that surface such risks early and consistently across projects and target profiles.

We motivate a data-to-vision approach to static analysis: instead of relying on handcrafted rules alone, we (i) parse a fixed code snapshot, (ii) build a unified program graph (CFG+DFG with buffer/guard annotations and edge weights), (iii) compute local and chain risk indicators, and (iv) rasterize informative subgraphs into multi-channel images with class labels (Stack, Heap, Off-by-one). This transformation makes complex program structure amenable to mature, data-efficient vision pipelines while preserving the determinism and auditability required in regulated settings.

2. Related works

Buffer overflows remain an ongoing issue even in well-established development pipelines. Despite notable improvements in defence techniques and coding best practices, real-world attacks still exploit subtle differences between what developers assume and the actual behaviour of code paths and runtime environments. The reviews highlight that classic overflows still happen in modern system components, libraries, and application software, and their circumvention is often achieved by changing the context, assembly configuration, or using code fragments where bounds checks are inconsistent with the real semantics of data copying and formatting [1].

The rise of deep learning for code analysis has redefined the representations crucial for both vulnerability detection and root-cause localisation. One of the pioneering works demonstrating the power of graph representations was Devign: it learned to identify vulnerable functions by combining program semantics in a graph with a GNN architecture. This research showed that “flat” features are less effective than graph dependencies that maintain the structure of control and data [2]. Further approaches involving pre-training models on code were proposed, such as VulBERTa, which focuses on simplified and practical preprocessing. This reduces the barriers to integrating such models into real pipelines without compromising quality on reference sets [3]. Already in these early works, the shift from “detection by indicators” to generalised features of graphs and tokens was outlined, making them better transferable between projects.

Meanwhile, the community has been actively investigating subject-specific graph and task configurations. In web and PHP environments, ideas for vulnerability detection have developed into models that integrate graphs with lexical signals and runtime context; it has been demonstrated that accurately encoding sources, sinks, and sanitisers in a graph can significantly reduce false positive noise [4]. At the binary analysis level, graph matching variants have emerged for identifying homologous vulnerabilities, where enhanced focus enables comparing fragments from different assemblies or optimisation profiles, while maintaining invariant templates of vulnerable structures [5]. At a more detailed level, property graphs have been proposed to describe programme entities and relationships in a unified way for precise localisation tasks, when it is necessary not only to “detect” the vulnerability class but also to produce a compact subgraph relevant to the cause of the error [6]. With the advent of LLMs, the natural progression was to incorporate the context of large language models into the CPG representation of code: such synthesis enhances portability and improves triage quality because LLMS effectively “fill in” gaps in local features and help reduce the number of false positives in poorly commented or non-canonically structured code [7].

The explainability issue, a key challenge for the practical deployment of SAST/ML detectors, is gradually being addressed through counterfactual justifications and local causality indicators. Counterfactual explanation methods for graph models enable us to identify which nodes or edges in a subgraph contributed to the model's decision; this facilitates targeted code correction by developers and allows tracking of regressions in subsequent commits [8]. At the same time, the quality of such explanations relies heavily on the accuracy of the data: active learning at the linear annotation level has demonstrated that systematic label noise in open sets can be mitigated by interactively reassessing “suspicious” examples and prioritising for review those that most influence the decision boundary [9]. The rise of micro-benchmarks for static analyzers and LLMs enables tools to be compared on stable, controlled tasks to identify subtle biases and environmental dependencies [10].

Meanwhile, studies on generalisability across projects and languages reveal train/validation/test leakage issues in some popular datasets and emphasise the need for rigorous, project-specific splits and consistency checks of results [11]. On the training side, example selection schemes are being actively studied: for instance, discarding "hard-to-learn" data at early stages helps stabilise the decision boundary and speeds up convergence without sacrificing quality on real-world problems [12]. Finally, cross-language datasets that include patch commits help link a vulnerability class to a specific remediation and also reduce the risks of overtraining to the stylistic patterns of individual repositories [13].

Classical static methods have not vanished but have been integrated into hybrid graph schemes. Heterogeneous graph models that simultaneously encode different types of nodes/edges (tokesemes, ASTs, CFGs, DFGs, library calls, contracts) demonstrate that a coherent representation of content and control dependencies enhances detection and localisation, especially for buffer operations [14]. In the web domain, there is a growing interest in LLM-based techniques that learn from mixed signals (code, templates, parameter tracking) and can identify invariants of checks and sanitisation in dynamically generated constructs [15]. A broader assessment of LLM approaches highlights both the prospects, such as generating correction hints and providing code assistance, and the threats, including hallucinations, sensitivity to hints, and instability with noisy data. It is especially important to keep the "human in the loop" and to maintain traceability of signal sources for auditing [16]. Parallel work involving counterfactual augmentation shows that artificial yet semantically consistent variants of vulnerable or safe fragments improve the distinction in borderline cases, particularly off-by-one and fencepost situations, where formal boundary checks conflict with the actual amount of copying [17].

The overall trend towards broadening languages and platforms has led to active investigation of "non-standard" environments. For example, a GNN detector has been proposed for Go, which considers the peculiarities of typing and patterns of the standard library, demonstrating competitive metrics within the domain. [18]. Systematic reviews compare different approaches and conclude that, without carefully designed datasets and reproducible pipelines, method comparisons are invalid. The environment, preprocessor settings, and feature building artefacts must be clearly documented to distinguish the "model input" from the "data input" [19]. At the code preprocessing stage, slicing methods are crucial: emphasising relevant slices around buffer operations significantly reduces noise and enhances root cause localisation, especially in large monolithic functions and interprocedural scenarios [20].

A separate layer concerns data quality and the example selection policy. Active learning and semi-automatic "relabeling" of complex cases help reduce systematic bias and leakage but require robust data traceability and audit trails at the project, patch, and build profile levels to prevent class blurring and leakage between training and test cases [9, 11, 12]. Micro-benchmarks complement these practices by enabling detailed comparison of tools and exposing implementation-detail dependencies that are hard to capture on large datasets [10], while cross-language corpora with commit pairs support realistic "before-and-after" remediation scenarios [13].

For the sake of completeness, it is worth mentioning related areas that focus not on code itself but on network and architectural aspects of cybersecurity. These areas illustrate important organisational and technical patterns relevant to CI/CD practice and risk modelling. In network security, a combination of passive DNS monitoring and active DNS probing has been proposed to detect botnets employing anti-evasion techniques. Two studies show that combining multiple surveillance channels (passive and active) enhances resistance to evasion and enables more accurate identification of malicious domain patterns [21, 22]. At the architecture level of multi-computer systems, a method and criteria for selecting the next centralisation option with traps and baits have been developed: this line of work demonstrates how formalised decision-making rules and comparison of alternatives influence overall cyber resilience and should be aligned with the goals of system protection and observability [23, 24]. Finally, corporate network cybersecurity assessment systems focus on integrated health indicators that combine metrics from different layers and enable the identification of "bottlenecks" in the infrastructure, which directly influences patch prioritisation

and deployment policies [25]. Although these studies use artifacts other than source code, their methodological logic—integrating different types of signals, formal decision criteria, and reproducibility auditing—is consistent with the "data-to-vision" approaches to SAST that we promote.

In summary, the current state of research on code vulnerability detection is shifting from "point" indicators to structurally consistent graph representations and data "cleansing" procedures, where solution explainability and pipeline reproducibility become just as important as absolute accuracy metrics. It is this approach—unification of program graphs, stabilisation of preprocessor profiles, data control, and project-level splits—that ensures SAST detection aligns with CI/CD practices and enables results to be transferred across projects, languages, and build configurations.

3 Methodology: reproducible data-preparation pipeline

We begin with a fixed snapshot of the repository S in a specific state (commit SHA or merge-ref) and a stabilised preprocessor profile. This removes nondeterminism caused by compilation conditions, macros, and assembly variants, ensuring that any subsequent feature development can be reproduced bitwise[15, 16]. After normalisation, the parser produces an inventory of programme entities that will be utilised at all stages of graph and risk indicator development:

$$I(S)=\{F,V,B,O\} \tag{1}$$

where S is a code snapshot, F is a set of functions, V is a set of variables, B is a set of buffers, and O is a set of memory operations. This inventory serves as the "single source of truth" for node and edge identifiers and enables tracing the origin of each feature back to a line of code [17, 18].

To combine structural and data dependencies, we represent the program as a unified graph [19, 20]. We explicitly preserve interprocedural calls/returns and read/write flows, since their interaction most often leads to overflows in real-world configurations:

$$G=(V,E),E=E^{CFG}\cup E^{DFG} \tag{2}$$

where V represents the vertices of operations, buffers, and call/return points, CFG are the control edges (including call/ret), DFG are the read/write data edges with attributes. Combining CFG and DFG provides a minimal but sufficient structure for risk assessment both locally and across execution paths.

Next, we define the effective capacity for buffer nodes. It differs from the "raw" size in that it accounts for overhead (e.g., null-termination of rows) and safety margins [20, 21]. This reduces the number of false positives when the formally allocated size does not equal the useful data capacity:

$$S_b(v)=cap(v)-overhead(v)-reserve(v) \tag{3}$$

where cap(v) is the allocated size, overhead(v) is the overhead (such as alignment, line termination, etc.), and reserve(v) is the reserved volume for security invariants. In practice, this means that even for obvious cases like char buff [16], the safe copy capacity is 15 bytes [22, 23, 24].

The local overflow criterion compares the estimated transfer size with the effective capacity of the receiving buffer.[25] We apply it only where sufficient cues are available to calculate the length (constants, format strings, loop invariants, or conservative upper bounds):

$$w(e)>S_b(dst(e)) \tag{4}$$

where w(e) is the estimated size in bytes for the write/copy edge, dst(e) is the receive buffer. When this criterion is satisfied, we mark the corresponding fragment as locally unsafe and include it in the candidate subgraphs for further analysis.

To obtain a continuous, differentiated threat assessment, we introduce a local risk metric. It correlates with the relative load but decreases with correct boundary checks and stack canaries, while increasing with off-by-one features and contextual factors:

$$R_{loc}(x)=\sigma\left(\alpha_1\frac{w(e)}{S_b}-\alpha_2 C(x)+\alpha_3 O_1(x)-\alpha_4 K(x)+\langle\beta, a(x)\rangle\right) \quad (5)$$

where σ (·) is sigmoid, α(x) is the vector of context attributes for the node or edge, C(x) is the presence of a correct boundary check, O1(x) is the off-by-one indicator, K(x) is the sign of an active stack canary, αi, β are weighting factors, $\frac{w(e)}{S_b}$ is the relative load. Thanks to this form, we can compare candidates by "threat strength" rather than just binary triggers.

Since overflows often result from a series of actions, we gather local contributions along the execution path. This provides a risk score for a particular data transfer journey from source to write point:

$$ChainRisk(\pi)=1-\prod_{x\in\pi}(1-R_{loc}(x)) \quad (6)$$

where π is the path in G, $R_{loc}(x)$ is the local risk of element x. The interpretation is simple: the product is the "safety probability" of the path; the complementary term $\prod(1-R_{loc})$ is the risk that at least one element along π will cause a problem.

After ranking the subgraphs by risk, we apply overflow-type labelling. The decision is guided by a class-specific utility that combines memory context features, allocation/copy and bounds checking signatures, and the off-by-one signal if it dominates:

$$label(x)=arg\max_{k\in \text{Stack},\text{Heap},\text{Off-by-one}}\left(\gamma_k \Phi_k(x)\right) \quad (7)$$

where $\Phi_k(x)$ is the vector of contextual features for class k, γ_k is the priority weights. This labelling is convenient for further training and evaluation, as it immediately reflects the practical categories of fixes in CI/CD.

At the sampling stage, it is important not to artificially "make things easier." Therefore, we emphasise "hard" negatives: these are subgraphs without a positive label, but with a high buffer load and clear protective signals. They reduce the model's tendency to confuse the absence of guards with the very presence of vulnerability:

$$N(x)=I[\frac{w(e)}{S_b}>\tau_1, C(x)\vee K(x), O_1(x)=0] \quad (8)$$

where τ_1 is the relative loading threshold, C(x) ∨ K(x) indicates the presence of at least one protective signal, and $O_1(x) = 0$ signifies the absence of an explicit off-by-one. During training, such examples enhance the detector's specificity, compelling it to depend on causal rather than superficial correlations, thereby offering a mathematical foundation without overwhelming the section.

Our prototype models graph-to-image change directly, without using a black box method. For each possible buffer action, we pull out a subgraph. This subgraph includes the buffer node, its copy/format action, and nearby control/data links. Then, we put this subgraph onto a set two-dimensional area using a set layout. Control-flow followers go on one side, data links on the other, and cross-program jumps are kept to a few layers. Nodes in the same grid spot are grouped by simple pooling. The resulting frame has many channels. Each channel shows node type (buffer, index, guard, copy call, math), edge type (control, data, call), and simple risk numbers from the graph (like index

and copy size limits, local w(e) guesses, and if guards exist). There are also simple channels for the target class tag (Stack, Heap, Off-by-one) for teaching and a mask of active areas.

These frames go into a normal one-stage detector like YOLO. Each frame is seen as an image with one or more weak objects, and the detector guesses bounding boxes and class tags over the grid. The teaching process is like standard object detection: it uses a mix of class loss and IoU-based regression loss. We adjust the confidence and IoU levels, so the detector can fit into CI/CD without giving too many bad alarms. It is key that all settings that change the image creation are in the same setup as the static analysis tools.

4 Implementation and reproducibility

The implementation is based on the principle of complete determinism: a single fixed code snapshot, a single fixed toolchain, and a single versioned data schema. The source code is always retrieved from a specific commit SHA or merge - ref. The working tree is checked for "dirt" before starting, and the preprocessor profile is stabilised and recorded in the manifest alongside the target ABI, a set of macros (DEBUG/RELEASE, RTOS flags), the language standard, and a comprehensive list of include paths. The parsing is carried out on top of Clang/LLVM with full preprocessing; the AST is stored in a standard form, on which a unified programme graph is created: control arcs (including interprocedural call/return) and data dependencies (read/write, def-use) with attributes for further evaluation of w(e). For heterogeneous assemblies, a compilation database is utilised; if it is not provided, it is reconstructed by intercepting the compilation process, after which “thin” shim headers eliminate random variations in system includes between distributions.

All intermediate representations have their own schema versions and immutable field semantics. The inventory {F, V, B, O} is serialised into inventory. Json with global identifiers and coordinates within the files. The graph and its attributes are serialized into a compact binary container based on protobuf. Risk indicators, including Rloc and Chain Risk are present in the risk.jsonl record stream with references to source nodes, edges, and preprocessor context. Training frame generation and markup occur after the graph stage; each artifact is accompanied by a sha256 hash and a provenance record that details the container version, commit, and execution time.

Reproducibility is secured through containerisation as an OCI image with fixed digest identifiers for the dependency chain. The tool operates under identical conditions on GitHub Actions, GitLab CI, and Jenkins without altering the build infrastructure: the input is always an unchanged snapshot, and the output remains the same set of artefacts and quality logs. Pseudo-random components (such as selection of "hard" negatives and tie-breakers during candidate subgraph conflicts) are governed by a single seed, which is activated in Python and C++ and stored in the startup file. Multithreaded stages are either unlocked or executed with a fixed number of workers and a stable task distribution, preventing race conditions when traversing large directory trees.

Quality control enforces invariants on the integrity of the AST/CFG, DFG balance, correctness of buffer attributes and the stability of w(e) estimates in response to changes in the order of file traversal. Any incompatible change in tool versions or preprocessor profile deliberately causes the run to enter an error state until the schema_version in the manifest is synchronously increased. This policy prevents hidden drift and guarantees that the results shown in the article are reproducible byte-for-byte across different environments and runtimes without needing illustrations or extra schemas.

The resulting dataset is moderately imbalanced, with much more safe code than code with off-by-one errors, which are not common. To fix this, we used a mix of basic methods. We reduced the number of easy safe code examples and increased the number of tricky error examples. We also used the risk scores in the graphs to add examples of safe code that looked risky but were okay. We did this because in CI/CD, it’s as vital to avoid false alarms as it is to catch actual errors. The system needs to tell apart truly bad code from code that just looks bad.

5 Case study

Consider a minimal example where overflow occurs only at a "thin" boundary, that is, when the length of the input string equals the buffer's capacity. The code demonstrates a classic fencepost problem, where formally available bounds checking does not ensure copy safety:

```
int copy_user(char *dst, size_t n, const char *src) {
    size_t len = strlen(src);
    if (len <= n) {          // error: should be len < n
      memcpy(dst, src, len + 1);
      return 0;
    }
    return -1;
}
```

At the inventory stage by (3.1) we have F = {copy_user}, V={dst,n,src,len}, B = {dst}, O = {memcpy}. The consistent graph of the program by (3.2) is contained in the CFG, the arcs from the function input to the if branches and to memcpy, and in the DFG, the edges src⟶memcpy. arg2, dst⟶memcpy. arg1, len⟶ (+1) ⟶memcpy. arg3, as well as the dependency on the predicate len <= n. This creates a subgraph in which the decision to copy is conditionally closed to the value len and the capacity parameter n.

The effective capacity for the receiver is modelled by S_b(dst). For interface functions with parameter n, it is natural to interpret cap(dst)≈n, and there is no explicit overhead at the level of memory allocation, and the reserve for invariants is zero. Therefore, S_b(dst) = n. The transfer volume w(e) for the edge corresponding to the memcpy call is defined as len + 1, since the null terminator is also copied. The local criterion works exactly for the case len=n, where w(e)=n+1>n=S_b(dst), which signals a guaranteed overflow.

The bounds check semantics in the predicate «len <= n» generate an off-by-one signal. The O_1 flag activates because the comparison permits equality, while the branch copies 'len + 1'. The safety flag C for this fragment is invalid (formally the check exists, but its logic does not align with the extent of copying), so it does not decrease the risk in the model; the K flag is zero because the stack canary does not influence the safety of the memcpy operation. In this setup, the local risk rises due to the ratio w(e)/Sb and active O1, which is not offset by C or K. Since the execution path from the predicate to the call is brief and lacks additional risk "dampers", the path estimate ChainRisk nearly matches the local one, and for len = n, it approaches unity.

The subgraph classification selects the Off-by-one category. The key feature here is the combination of a fencepost predicate with copying, which explicitly increments the length by one. For len<n, the risk decreases and the subgraph is more likely to receive a neutral label; for len>n, the situation is no longer “on the edge” and indicates a typical overflow. However, this branch is not executed because of the if statement.

Regarding the “frame” X, this example is interpreted through touch rather than sight: in the load channel, a zone around the memcpy node is highlighted; in the off-by-one channel, an active signal appears in the predicate-argument-copy cluster; in the protection channel, there is no contribution from the useful boundary-check; and in the local risk channel, a “hot” maximum is formed. The annotation Y in this case corresponds to a rectangle covering the subgraph { predicate len <=n, computation len + 1, memcpy node } and the Off-by-one class.

This short example shows how a boundary parameter is managed in many interfaces, where it's viewed as a capacity and a null terminator gets added automatically. Such patterns show up in our data in both library helpers and in wrappers made around safer APIs. This makes them a key source of examples. The subgraph's compactness means it can be used as a visual unit test for the graph-to-image change. Any encoding change that hides the off-by-one signal quickly makes the detector perform worse on this case.

Table 1
Overall detection on a project-wise split

Method	Precision	Recall	F1-measure	Time (s/file)
Cppcheck (rules)	0.34	0.62	0.44	1.8
Flawfinder (rules)	0.28	0.57	0.38	1.5
YOLO (tokens only; no graphs)	0.79	0.82	0.80	3.9
Ours (graphs + risk)	0.95	0.95	0.946	8.7

During training, this fragment presents a challenge because external checks can mislead simple rule templates into classifying it as safe. Despite this, we include it as a positive example. After adjusting the copy length or predicate, we also use it to generate paired hard negatives. This ensures that the detector prioritizes the causal combination of w(e), guard conditions, and off-by-one cues, instead of just detecting the presence of a check or copy call.

Our results are intentionally limited to baselines that integrate into a similar CI/CD setup and are assessed using the same project split. Graph-based neural models, like Devign and pre-trained language models like VulBERTa, show good F1 scores on their own benchmarks, usually between 0.6 and 0.9 based on the dataset and task setup. Still, they are usually assessed on specific vulnerability datasets (Devign, Draper, REVEAL) that use different labeling and splitting methods. So, we don't put their published numbers in Table 1. We think of them as separate methods. Our work centers on a repeatable graph-to-image data method, which can be a data source for these models in the future. The current tests only compare tools that we can run with the same resource and setup limits.

6 Discussion and limitations

The proposed pipeline addresses the practical issue of reproducibility in SAST: all stages — from capturing a code snapshot to generating graphs and markup — are deterministically controlled by a single preprocessor profile, tool versions, and data schemas, which makes the results stable for CI/CD environments and suitable for auditing. However, there are methodological limitations as well. First, the estimate of w(e) inevitably depends on conservative upper bounds and partial symbolic analysis; in the presence of complex macros, inline assembler, "thin" library functions, or platform-dependent behaviour, we either override conservative estimates or mark edges as undefined to avoid "inventing" the accuracy. Second, the overflow landscape depends on the build configurations: different preprocessor profiles can activate incompatible code branches, leading to multiple alternative graphs for the same repository; we address this by executing separate runs for each profile, though this increases computational cost. Third, risk indicators — although formally defined — rely on the quality of the input features (types, sizes, loop invariants) and are therefore vulnerable to incompleteness or noise. Class marking is based on context rules and can be easily applied to “canonical” overflow patterns, but in different environments (such as specific RTOS, non-standard allocators, or generated code), it is necessary to adapt guard detectors and recalculate effective capacity. Finally, the method intentionally does not address dynamic runtime effects, as it relies solely on static information; for such cases, hybrid SAST-DAST methods or targeted fuzzing on “hot” subgraphs are needed.

Beyond these methodological points, there are threats to validity and scalability that matter in production. The precision-recall trade-off in the pipeline depends on thresholding of risk indicators and class-specific heads; mis-calibration across repositories may inflate false alarms in large

monorepos or underreport rare off-by-one cases. Dataset shifts and annotation bias can subtly steer models toward spurious correlations; ablation and differential-testing against profile variants helps detect such drift but adds compute cost. Containerization mitigates environment drift, but evolving third-party headers or transitive build tools can break reproducibility unless SBOM pinning and digest-locked mirrors are enforced. Finally, in CI/CD integration, latency and coverage must be balanced: even with caching and incremental parsing, full graph extraction and rendering can tax shared runners; a practical mitigation is staged evaluation (fast pre-filter, then deep analysis on "hot" subgraphs), combined with human-in-the-loop triage for borderline findings and periodic re-calibration of blocking thresholds.

7 Conclusion and future work

We present a reproducible data preparation pipeline for detecting buffer overflows in C/C++: a code snapshot with fixed profiles, a unified programme graph with transfer weights, formal indicators of local and path risks, and consistent class labels. This “data-to-vision” transformation makes the programme dependency structure suitable for further detection without sacrificing audit transparency. All figures are derived from reproducible artefacts and can be verified afterwards.

Further work involves enhancing the accuracy of w(e) estimation—adding deeper symbolism for cycles and format strings, verifying length constraints—expanding guard detection with support for library and platform-specific contracts, and implementing automatic triage of candidate subgraphs with human oversight to minimise false positives during initial integrations. A separate approach involves multi-profile analysis (using multiple preprocessor configurations for the same commit) with intelligent merging of risk signals, as well as creating reference datasets with strict control to prevent leaks between train, validation, and test sets at both project and patch levels. Practically, we intend to publish a replication package containing a container, manifests, and control runs in open repositories to promote independent verification and further comparison.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] M. A. Butt, Z. Ajmal, Z. I. Khan, M. Idrees, Y. Javed, An in-depth survey of bypassing buffer overflow mitigation techniques, Applied Sciences 12(13) (2022) 6702. doi:10.3390/app12136702.

[2] Z. Zhao, M. Xu, Y. Zhang, L. Chen, X. Luo, Y. Deng, et al., GMN+: A binary homologous vulnerability detection method based on graph matching neural network with enhanced attention, Applied Sciences 14(22) (2024) 10762. doi:10.3390/app142210762.

[3] H. Hanif, S. Maffeis, VulBERTa: Simplified source code pre-training for vulnerability detection, Proc. Int. Joint Conf. on Neural Networks (IJCNN 2022), IEEE (2022). doi:10.1109/IJCNN55064.2022.9892280.

[4] C. Lin, Y. Xu, Y. Fang, Z. Liu, VulEye: A novel graph neural network vulnerability detection approach for PHP application, Applied Sciences 13(2) (2023) 825. doi:10.3390/app13020825.

[5] Y. Tian, S. Chen, F. Yin, W. Zhou, CrossVul: A cross-language vulnerability dataset with commit data, Proc. ACM Joint Eur. Softw. Eng. Conf. Symp. on the Foundations of Software Engineering (ESEC/FSE 2021), ACM (2021). doi:10.1145/3468264.3473122.

[6] M. Shao, Y. Ding, J. Cao, Y. Li, GraphFVD: Property graph-based fine-grained vulnerability detection, Comput. Secur. 151 (2025) 104350. doi:10.1016/j.cose.2025.104350.

[7] A. Lekssays, H. Mouhcine, K. Tran, T. Yu, I. Khalil, LLMxCPG: Context-aware vulnerability detection through code property graph-guided large language models, arXiv preprint arXiv:2507.16585 (2025).

[8] Z. Chu, Y. Wan, Q. Li, Y. Wu, H. Zhang, Y. Sui, G. Xu, H. Jin, Graph neural networks for vulnerability detection: A counterfactual explanation, Proc. ACM SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA 2024) (2024) 1–13. doi:10.1145/3650212.3652136.

[9] A. Kallingal Joshy, M. S. Alam, S. Sharmin, Q. Li, W. Le, ActiveClean: Generating line-level vulnerability data via active learning, arXiv preprint arXiv:2312.01588 (2023).

[10] R. A. Dubniczky, K. Z. Horvát, T. Bisztray, M. A. Ferrag, L. C. Cordeiro, N. Tihanyi, CASTLE: Benchmarking dataset for static code analyzers and LLMs towards CWE detection, arXiv preprint arXiv:2503.09433 (2025).

[11] R. Rahimi, M. Shimmi, H. Okhravi, Data and context matter: Towards generalizing AI-based software vulnerability detection, arXiv preprint arXiv:2508.16625 (2025).

[12] X. Lan, T. Menzies, B. Xu, Smart Cuts: Enhance active learning for vulnerability detection by pruning hard-to-learn data, arXiv preprint arXiv:2506.20444 (2025).

[13] G. P. Bhandari, A. Naseer, L. Moonen, CVEfixes: Automated collection of vulnerabilities and their fixes from open-source software, Proc. Int. Conf. on Predictive Models and Data Analytics in Software Engineering (PROMISE 2021), ACM (2021). doi:10.1145/3475960.3475985.

[14] Z. Song, J. Wang, S. Liu, Z. Fang, K. Yang, HGVul: A code vulnerability detection method based on heterogeneous source-level intermediate representation, Security and Communication Networks (2022) 1919907. doi:10.1155/2022/1919907.

[15] D. Cao, Y. Liao, X. Shang, RealVul: Can we detect vulnerabilities in web applications with large language models?, arXiv preprint arXiv:2410.07573 (2024).

[16] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray, Y. Chen, Vulnerability detection with code language models: How far are we?, arXiv preprint arXiv:2403.18624 (2024).

[17] D. Egea, B. Halder, S. Dutta, VISION: Robust and interpretable code vulnerability detection leveraging counterfactual augmentation, Proc. AAAI/ACM Conf. on AI, Ethics, and Society 8(1) (2025) 812–823. doi:10.1609/aies.v8i1.36592.

[18] L. Yuan, Y. Fang, Q. Zhang, Z. Liu, Y. Xu, Go source code vulnerability detection method based on graph neural network, Applied Sciences 15(12) (2025) 6524. doi:10.3390/app15126524.

[19] M. Shimmi, H. Okhravi, R. Rahimi, AI-based software vulnerability detection: A systematic literature review (2018–2023), arXiv preprint arXiv:2506.10280 (2025).

[20] S. Salimi, M. Kharrazi, VulSlicer: Vulnerability detection through code slicing, J. Syst. Softw. 193 (2022) 111450. doi:10.1016/j.jss.2022.111450.

[21] O. Savenko, S. Lysenko, A. Kryschuk, Multi-agent based approach of botnet detection in computer systems, CCIS, 291 (2012) 171–180. https://doi.org/10.1007/978-3-642-31217-5_19 .

[22] O. Pomorova, O. Savenko, S. Lysenko, A. Kryshchuk, Multi-Agent Based Approach for Botnet Detection in a Corporate Area Network Using Fuzzy Logic, Communications in Computer and Information Science, 370 (2013) 243-254, ISSN: 1865-0929. https://doi.org/10.1007/978-3-642-38865-1_16.

[23] O. Pomorova, O. Savenko, S. Lysenko, A. Kryshchuk, K. Bobrovnikova, A technique for the botnet detection based on DNS-traffic analysis, in Proc. 22nd Int. Conf. Computer Networks, Brunów, Poland (2015) 127–138.

[24] S. Lysenko, O. Pomorova, O. Savenko, A. Kryshchuk and K. Bobrovnikova, DNS-based Anti-evasion Technique for Botnets Detection, in Proceedings of the 8-th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, Warsaw (Poland), September 24–26, 2015. Warsaw. Pp. 453–458.

[25] I. Ramskyi, A. Drozd, O. Lyhun, O. Ponochnova, System for cybersecurity evaluation of corporate networks, Computer Systems and Information Technologies 2 (2025) 123–131. doi:10.31891/csit-2025-2-14.